Security Lab – Hacking-Lab Challenges Part 2

Introduction

In this lab, you'll do several Hacking-Lab challenges in the context of SQL injection and XML External Entity injection. The goal is to better understand and gain practical experience with such attacks. To access the challenges, select event *Challenges Part 2* in the Hacking-Lab – this shows you the five challenges that are part of this event. It's recommended to work through the challenges in the given order.

Please solve the challenges on SQL injection manually, i.e., without using *sqlmap*. This is important to achieve a real learning benefit. Once you have completely solved a challenge, feel free to check whether it can also be solved using *sqlmap*.

1 SCHOGGI: SQL Injection on Login Form

1.1 Goal

Get access as *the user that corresponds to the first row* in the user database by exploiting an SQL injection vulnerability in the login functionality. Once you have achieved this, get access as an *admin*. Assume that you neither know existing usernames nor passwords, so you are not allowed to use such information to achieve the attack goals.

1.2 Required solution for full points

- A description of the attacks you used to first get access as the user that corresponds to the first row in the user database and then to get access as an *admin* (include the injection strings you used).
- The username of the user that corresponds to the first row in the user database (you can get this by clicking *PROFILE* after having logged in as this first user) and the usernames of all admins (you can get them by clicking *MANAGE USERS* after having logged in as an *admin*).

1.3 Hints

- The used database schema is MySQL. In MySQL, comments can either be started with -- or #. If you use --, then this must be followed by a space character.
- Assume that a SELECT statement that uses the entered username and password is used for the login (similar as described in the lecture video / slides), and that the first row that is returned by this SELECT statement (if any) is used to identify the user. Note that when trying to do an SQL injection attack, it's always helpful to write down the (assumed) SQL statement and how the user-provided values may be used in this statement, because this makes it easier to come up with a working attack string.
- The Chocoshop uses hashed passwords. Assume that the received password is first hashed (in some way, the actual method is not relevant in this challenge), and this hash is then used in the SELECT statement, together with the username. This has a direct impact on the attack strategy you should use and whether you should use the username field or the password field (or both) to do the actual attack.
- The actual login request uses a REST API call. Use, e.g., the Firefox *Web Developer Tools* or *Burp* to inspect requests and responses.
- To get access as *admin*: Use the SQL *LIMIT* keyword to make sure the SELECT statement returns a specific row. For instance, *SELECT * FROM Product LIMIT offset,row_count* returns *row_count* rows starting at row *offset* (the offset of the first row is 0). Therefore, as a specific example, *SELECT * FROM Product LIMIT 2,1* returns the 3rd row in table *Product*.

- And a final hint: The web application only processes usernames up to a certain length. Therefore, especially when trying to log in as an *admin*, make sure your attack uses only as many characters as needed.

2 SCHOGGI: Union-Based SQL Injection

2.1 Goal

Get the usernames, password hashes and credit card numbers of all users by exploiting an SQL injection vulnerability in the *Search by product name* functionality of the shop.

2.2 Required solution for full points

- A description of the attack steps you used to get the usernames, password hashes and credit card numbers of all users (include the injection strings you used).
- The password hash and the credit card number of user *charlie*.

2.3 Hints

- The actual search request uses a REST API call. Use, e.g., the Firefox *Web Developer Tools* or *Burp* to inspect requests and responses.
- After you have solved this challenge manually (don't do it beforehand!), it's recommended to check whether it also works with *sqlmap*. Copy the search request (when using, e.g., *test* as a search string) into a text file (e.g., *request.txt*) and use *sqlmap* as described in the lecture videos / slides. Using the additional option *--force-ssl* makes sure *sqlmap* uses HTTPS (per default, it uses HTTP).

3 SCHOGGI: Blind SQL Injection (SQLI)

3.1 Goal

Get the credit card number of user *mallory* by exploiting a blind SQL injection vulnerability in the *Login* functionality.

3.2 Required solution for full points

- A description of the attack steps you used to get the credit card number of user *mallory* (include the injection strings you used).
- The first four characters of the credit card number of user *mallory*.
- Optional: Write a piece of code that automates the attack to get the complete credit card number (without using *sqlmap*).

3.3 Hints

- Usernames and credit card numbers are stored in columns *username* and *credit_card* in table *users*. Note that the attack would also be possible without this information, but then you'd first have to get this information from the INFORMATION_SCHEMA schema, by using basically the same attack you are going to use in this challenge.
- First, try out existing (e.g., *alice*) and non-existing (e.g., *test*) usernames in the login form and observe the behavior. The web application tells you *Wrong username or password* in both cases. However, looking at the actual login requests and responses, one can see the response is *Invalid user or password* with an existing username and *Error during login* otherwise.
- Let's assume there's an SQL injection vulnerability involved here. As the web application only returns these two different texts, the previously used approach based on UNION SELECT most likely does not work to get additional data such as the credit card number of *mallory*. However, by using so-called *Blind SQL Injection*, you may be able to inject different SELECT statements, and

depending on the text you get in the response, learn something about the result of the injected statement.

- Thinking about the behavior observed when submitting usernames, it may be that the received username is used in a SELECT statement to get the corresponding row from the database, e.g., *SELECT * from users WHERE username='alice'* if *alice* is used as username. In a second step, the password may be checked using this row, but we don't care about this second step in this attack.
- From now on, it's best to use the login request in the *Repeater* component of Burp, as this allows you to easily play with different usernames and directly see the response.
- To verify the existence of an SQL injection vulnerability, use *alice'--* as username (don't forget the space character after --). Without an SQL injection vulnerability, one would expect *Error during login*, as the username *alice'--* certainly does not exist. However, the response contains *Invalid user or password*. The only reasonable explanation for this is that there's indeed a vulnerability in the sense that string concatenation is used to build the SELECT statement, and that the resulting SELECT statement is something like *SELECT * from users WHERE username='alice'-- '*. This is a valid SELECT statement to get the row of *alice* (remember that *-- '* is treated as SQL comment, so we can ignore it).
- Next, try *alice' AND (1=1)--* . This also results in *Invalid user or password*. Likewise, try *alice' AND (1=0)--* , which results in *Error during login*. Make sure you truly understand why these injection strings for the username result in the two different responses. If you don't understand this, write down the resulting (assumed) SELECT statements; this should help to make things clear. Note that the parentheses wouldn't be needed, but it's nerveless a good idea to use them to prevent syntax problems during the following steps.
- What we have now is a so-called *oracle*: You can use any SQL expression instead of *1=1* (TRUE) or *1=0* (FALSE) and the application tells us whether this SQL expression is TRUE (where you get *Invalid user or password*) or FALSE (*Error during login*).
- This allows you to learn the credit card number of user *mallory*, character by character, from left to right. To do this, replace *1=1* from above with a SELECT statement that returns TRUE if you guess the first character of the credit card number correctly and FALSE otherwise (you have to go through all characters 0-9 – one of them is the correct first character and only this one should return *Invalid user or password*). Once you have found the first character, you can then get the second character in the same fashion and so on. Note that the credit card number consist of characters 0-9 and the space character.
- The first character of the credit card number of *mallory* is *2*. This information may help you to verify that the SELECT statement you are using in the step above indeed allows you to find out the correct character.
- After you have solved this challenge manually (don't do it beforehand!), it's recommended to check whether it also works with *sqlmap*. Copy the login request (when using *alice* as username and, e.g., *test* as password) into a text file (e.g., *request.txt*) and call *sqlmap* as described in the lecture videos / slides. Using the additional option *--force-ssl* makes sure *sqlmap* uses HTTPS (per default, it uses HTTP) and using *--technique=B* makes sure that *sqlmap* only uses blind SQL injection techniques, which significantly speeds up the process:

4 SCHOGGI: XML External Entity (XXE)

4.1 Goal

Get the content of */etc/shadow* by exploiting an XML External Entity injection vulnerability in the *Bulk Order* functionality.

4.2 Required solution for full points

- A description of the attack steps you used to get the content of */etc/shadow*.

- A screenshot that shows the content of */etc/shadow* as delivered by the web application.

4.3 Hints

- To get to the *Bulk Order* functionality, log in as *alice*/*alice.123* and click *BULK ORDER* at the top. There, you can also download a sample XML file as a basis for the attack.

5 SCHOGGI: Server-Side Request Forgery (SSRF) with XXE

5.1 Goal

Access a hidden endpoint on the Chocoshop web server by performing a Server-Side Request Forgery (SSRF) attack based on the same XML External Entity injection vulnerability in the *Bulk Order* functionality as in the previous challenge. SSRF is an attack where a server makes a request on behalf of the attacker to access a resource (e.g., on the internal network or on the web server itself) that isn't directly accessible by the attacker. Successfully accessing the hidden endpoint will reveal sensitive information such as the MySQL password and the JSON Web Token (JWT) signing key (note that JWTs will be discussed in detail later in the module SWS1).

5.2 Required solution for full points

- A description of the attack steps you used to get access to the hidden endpoint.
- A screenshot that shows the response (including the MySQL password and JWT signing key) as delivered by the web application.

5.3 Hints

- First, try to find the hidden endpoint on the Chocoshop web server. You could try to guess typical names, but there's a nice tool that can help you: *gobuster*. To search for common files and directories in a web application / on a web server that is reached at *https://www.site.com*, use it as follows in a terminal (option *-w* specifies the file with the directories/files that should be tried):
  gobuster dir -e -u https://www.site.com -w /usr/share/wordlists/dirb/common.txt
  Look out for a resource that seems to be existing, but that cannot be accessed.
- Access this resource with the browser, which reveals from where the resource can be accessed. This information tells you that accessing it using the XXE injection vulnerability you used before vulnerability should be possible.
- Assume the hidden endpoint is accessible on port 8888 on the web server (without this knowledge, you'd have to guess the correct port).

Lab Points

In this lab, you can get **2 Lab Points**. To get them, you must achieve all 1'000 Hacking-Lab points that you can get from solving the challenges in this lab.